Learning to program a real high-speed internet router.

Zhen Yi Pan

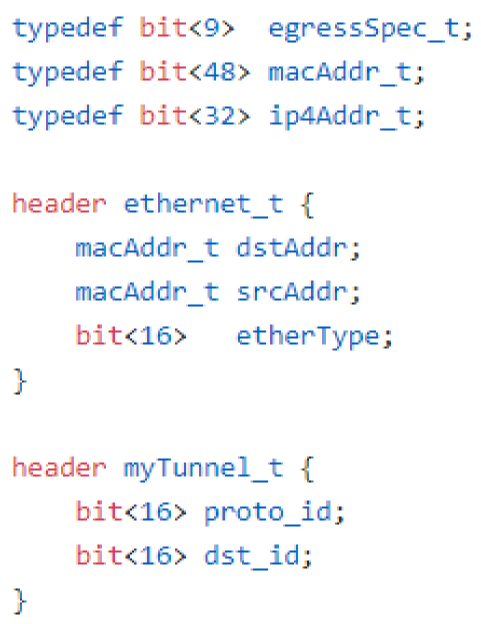
Abstract.
This paper is a summary of the work I did during the 2020 DIMACS REU program. This includes information about the P4 programming language and the Barefoot SDE. Information about the P4 programming language includes implementing exercises that exhibit network-level use cases of programmable switches through P4 programs. Information about the Barefoot SDE includes experiences with a real hardware device that can process data at 6.4 Tbit/s.

1. INTRODUCTION.

Internet traffic goes through a series of devices called routers. Routers are high-speed devices, meant to redirect packets of data quickly from one place to another. In traditional networks, the router's functions are implemented at the hardware level. This gives a performance boost to the speed of the router; however, this limits the device in terms of the actions that can be executed. Software-defined networking is defined by the decoupling of control and forwarding planes in the network. This separation enables the flexibility for the switch to exploit open communication protocols like OpenFlow. This allows operators to manage the entire network and its devices with code consistently, regardless of the underlying hardware, which may be from multiple vendors. My role is to understand the specific details of programmable switches, explore this emerging field in computer networking, and ultimately to work towards the larger research effort intended to define mechanisms and policies that carve out university network resources to various users by programming routers appropriately.

2. BACKGROUND.

P4 is a programming language for controlling how packets are processed by the forwarding plane in networking devices, such as routers and switches. Through the duration of the summer research, 2 different Github tutorials [3, 4] were used to learn different networking protocols and the P4 programming language.



```
typedef bit<9>  egressSpec_t;
typedef bit<48> macAddr_t;
typedef bit<32> ip4Addr_t;

header ethernet_t {
    macAddr_t dstAddr;
    macAddr_t srcAddr;
    bit<16>   etherType;
}

header myTunnel_t {
    bit<16> proto_id;
    bit<16> dst_id;
}
```

Figure 1. Example header section.

2.1 P4 language structure.

The first thing to notice when working through the early exercises is the unique structure of the language. The code is formatted into very specific sections including the headers, parser, checksum verification, ingress processing, egress processing, checksum computation, deparser, and the switch. Within these subsections, there was more formatting. Take the header shown in Figure 1 for example. At the very top, the figure contains type definitions with a name attached to a set number of bits. The image also contains example definitions of headers. The ethernet header is defined as an “ethernet_t,” which contains a destination address and source address using the predefined type, “macAddr_t.” There is also another field, “etherType,” defined directly with the number of bits, which in this case is 16. Nonstandard headers can be defined as well, such as “myTunnel_t” with specified bits defined as needed. These definitions follow a straightforward structure, which gives the code a neat and clean look. This nice formatting allows the code to be easily understood and this is consistent throughout the rest of the code body.

3. P4 EXERCISES.

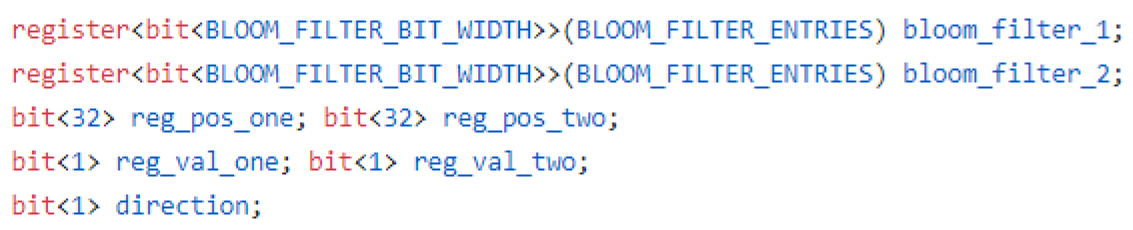
The Github tutorials [3, 4] explored many important concepts in computer networking, revealing vast complexities that go into a single program. A network is much more complicated than simply moving a packet from an arbitrary input port to an arbitrary output port of a router. There are many intricate aspects of the P4 programs that come together to make a working network as a whole.

3.1 Tunneling

The first concept is tunneling. Figure 1 presents the custom header, “myTunnel_t.” This can be used to determine if an incoming packet is of the IPv4 internet protocol or the IPv6 internet protocol. If the underlying infrastructure does not support IPv6, encapsulation of the IPv6 packet in an IPv4 packet allows the movement of data through such devices.

3.2 Firewall with bloom filters

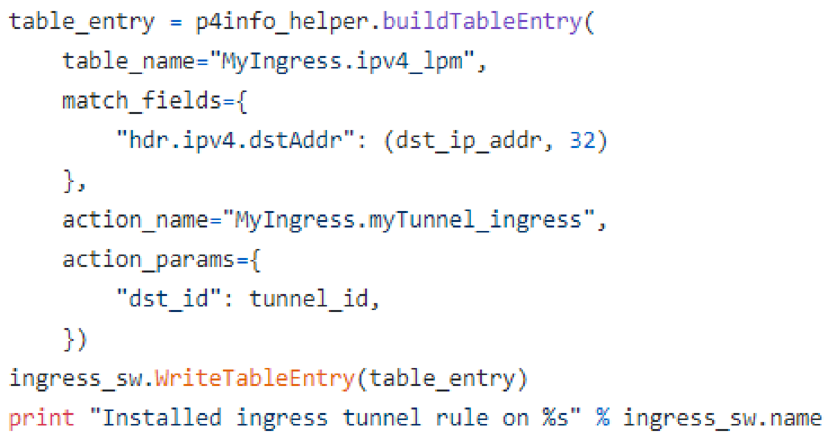
Another exercise introduced a kind of probabilistic data structure — the bloom filter.



```
register<bit<BLOOM_FILTER_BIT_WIDTH>>(BLOOM_FILTER_ENTRIES) bloom_filter_1;
register<bit<BLOOM_FILTER_BIT_WIDTH>>(BLOOM_FILTER_ENTRIES) bloom_filter_2;
bit<32> reg_pos_one; bit<32> reg_pos_two;
bit<1> reg_val_one; bit<1> reg_val_two;
bit<1> direction;
```

Figure 2. Example of a bloom filter in the P4 language.

A bloom filter is a data structure that maintains set membership with sublinear space on the number of items in a streaming setting. In the P4 language, it can be implemented with an array of registers. Bloom filters are a useful tool for speeding up and simplifying networking protocols. In the example shown in Figure 2, the bloom filter is used to construct a firewall. Chosen identifying fields (IPv4 source address, IPv4 destination address, TCP source port, and TCP destination port) of the packet are hashed using a predefined hash algorithm, and the return value will act as the index in the bloom filter. The value at the computed index will be set to 1 from the original value, which was 0. This setting of the value in the bloom filter will be completed when a packet is exiting the router. When another packet enters, the hash of the same fields (IPv4 source address, IPv4 destination address, TCP source port, and TCP destination port) will be computed to acquire an index and checked with the existing bloom filter to determine if the packet should be allowed or dropped. The bloom filter is a probabilistic data structure, which means there can be false positives, but there can never be false negatives. In the case of firewalls, the bloom filter may allow some packets that have hashed values overlapping with existing packets' hashed values to pass through. The bloom filter will never allow packets whose hashed values do not overlap with existing hashed packets' values. For example, if the hash algorithm computes an index with the value, 0, at the index, then the packet will not pass through.



```
table_entry = p4info_helper.buildTableEntry(
    table_name="MyIngress.ipv4_lpm",
    match_fields={
        "hdr.ipv4.dstAddr": (dst_ip_addr, 32)
    },
    action_name="MyIngress.myTunnel_ingress",
    action_params={
        "dst_id": tunnel_id,
    })
ingress_sw.WriteTableEntry(table_entry)
print "Installed ingress tunnel rule on %s" % ingress_sw.name
```

Figure 3. Python control plane.

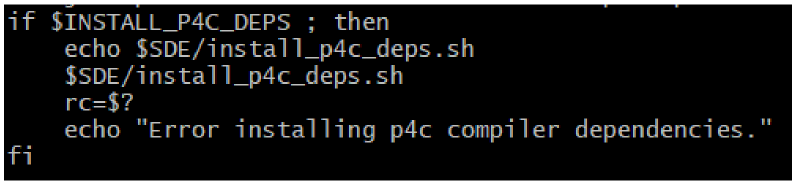
3.3 P4 control plane.

The P4 control plane exercise on Github [3] demonstrates how to populate the forwarding plane's table using Python. Figure 3 is a snippet of the control plane code taken from the p4runtime example, which populates the ipv4_lpm table with match fields, action names, and action parameters.

3.3.1 Explaining the relationship between the forwarding plane and the control plane.

An important part of the learning process is to understand the relationship between the forwarding plane and the control plane. The forwarding plane sets up the table and the table structure, while the control plane populates the table with table entries. An example of the forwarding plane would be to imagine an empty table with headers, "key," and "actions." The forwarding plane creates rules that say the "key" column of the table will take IPv4 destination addresses, as an example. It will also create rules that say the "actions" column will take only one of the three actions (ipv4_exact, drop, NoAction, for example) that were previously defined by you. These rules set up a contract for the control plane. The control plane must populate the "key" column of the table with IPv4 addresses, and not IPv6 addresses or MAC addresses. Similarly, the "actions" column must take either ipv4_exact, drop, or NoAction. It cannot take another action like ipv4_lpm, which was not defined in the forwarding plane table. All of the table information would be stored in a dedicated memory.

In learning P4, the 2 sets of Github tutorial were both exceptional tools to not only learn the P4 language, but also dive into various concepts and complexities of computer networking.



```
if $INSTALL_P4C_DEPS ; then
    echo $SDE/install_p4c_deps.sh
    $SDE/install_p4c_deps.sh
    rc=$?
    echo "Error installing p4c compiler dependencies."
fi
```

Figure 4. Error message in the install script.

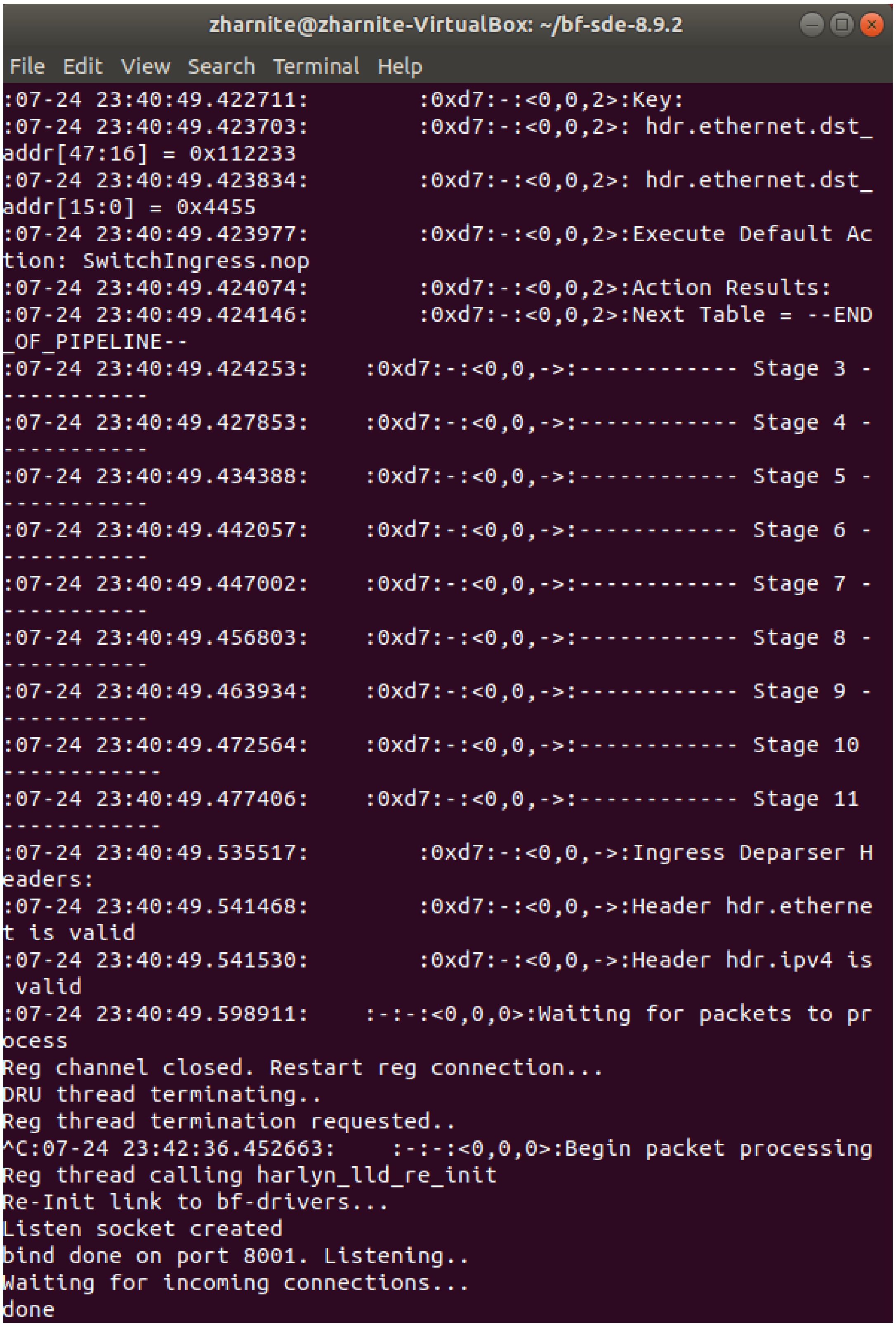
4. INSTALLING THE BAREFOOT SDE.

The Barefoot SDE is the software package used to interact with the Tofino switch hardware. It can be set up on a virtual machine and run with the tofino-model. Many problems occurred while setting the SDE up on both the virtual machine and the physical hardware.

4.1 Installation on the virtual machine.

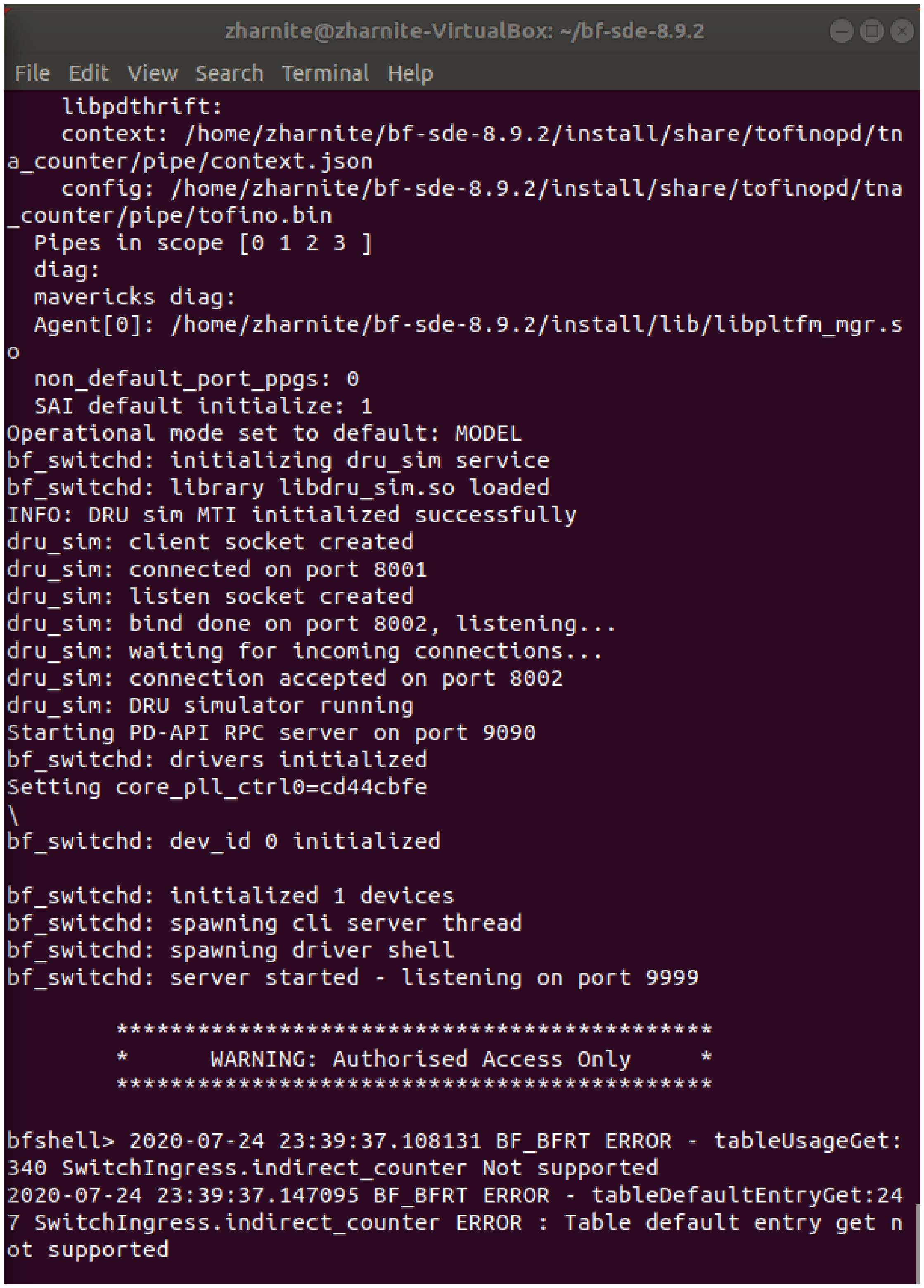
An error occurred in part one of the installation regarding the software dependencies. The install script had printed an error message no matter the status of the installation. With a thorough look through the script, one can encounter something rather evil displayed. This is shown in Figure 4. It is a message stating, "Error installing p4c installer dependencies." This may seem reasonable; however, at a closer look, it appears that after "$SDE/install_p4c_deps.sh" is run, the exit code is stored in "rc" with "rc=$?." The exit code in rc is not used again to determine if the error message should be printed. Thus, no matter the status of "$SDE/install_p4c_deps.sh," an error message will always print. Evil.

After plowing through the installation errors, a working environment was up on a Ubuntu 18.04 virtual machine. A few test cases were run on the tofino-model. Figure 5, Figure 6, and Figure 7 displays a working example of tna_counter's tofino-model, bfshell control plane, and Python test cases respectively.



```
zharnite@zharnite-VirtualBox: ~/bf-sde-8.9.2
File  Edit  View  Search  Terminal  Help
:07-24 23:40:49.422711:        :0xd7:-:<0,0,2>:Key:
:07-24 23:40:49.423703:        :0xd7:-:<0,0,2>: hdr.ethernet.dst_
addr[47:16] = 0x112233
:07-24 23:40:49.423834:        :0xd7:-:<0,0,2>: hdr.ethernet.dst_
addr[15:0] = 0x4455
:07-24 23:40:49.423977:        :0xd7:-:<0,0,2>:Execute Default Ac
tion: SwitchIngress.nop
:07-24 23:40:49.424074:        :0xd7:-:<0,0,2>:Action Results:
:07-24 23:40:49.424146:        :0xd7:-:<0,0,2>:Next Table = --END
_OF_PIPELINE--
:07-24 23:40:49.424253:    :0xd7:-:<0,0,->:------------ Stage 3 -
-----------
:07-24 23:40:49.427853:    :0xd7:-:<0,0,->:------------ Stage 4 -
-----------
:07-24 23:40:49.434388:    :0xd7:-:<0,0,->:------------ Stage 5 -
-----------
:07-24 23:40:49.442057:    :0xd7:-:<0,0,->:------------ Stage 6 -
-----------
:07-24 23:40:49.447002:    :0xd7:-:<0,0,->:------------ Stage 7 -
-----------
:07-24 23:40:49.456803:    :0xd7:-:<0,0,->:------------ Stage 8 -
-----------
:07-24 23:40:49.463934:    :0xd7:-:<0,0,->:------------ Stage 9 -
-----------
:07-24 23:40:49.472564:    :0xd7:-:<0,0,->:------------ Stage 10
------------
:07-24 23:40:49.477406:    :0xd7:-:<0,0,->:------------ Stage 11
------------
:07-24 23:40:49.535517:        :0xd7:-:<0,0,->:Ingress Deparser H
eaders:
:07-24 23:40:49.541468:        :0xd7:-:<0,0,->:Header hdr.etherne
t is valid
:07-24 23:40:49.541530:        :0xd7:-:<0,0,->:Header hdr.ipv4 is
 valid
:07-24 23:40:49.598911:    :-:-:<0,0,0>:Waiting for packets to pr
ocess
Reg channel closed. Restart reg connection...
DRU thread terminating..
Reg thread termination requested..
^C:07-24 23:42:36.452663:    :-:-:<0,0,0>:Begin packet processing
Reg thread calling harlyn_lld_re_init
Re-Init link to bf-drivers...
Listen socket created
bind done on port 8001. Listening..
Waiting for incoming connections...
done
```

Figure 5. Tofino-model on the virtual machine.


zharnite@zharnite-VirtualBox: ~/bf-sde-8.9.2
File Edit View Search Terminal Help

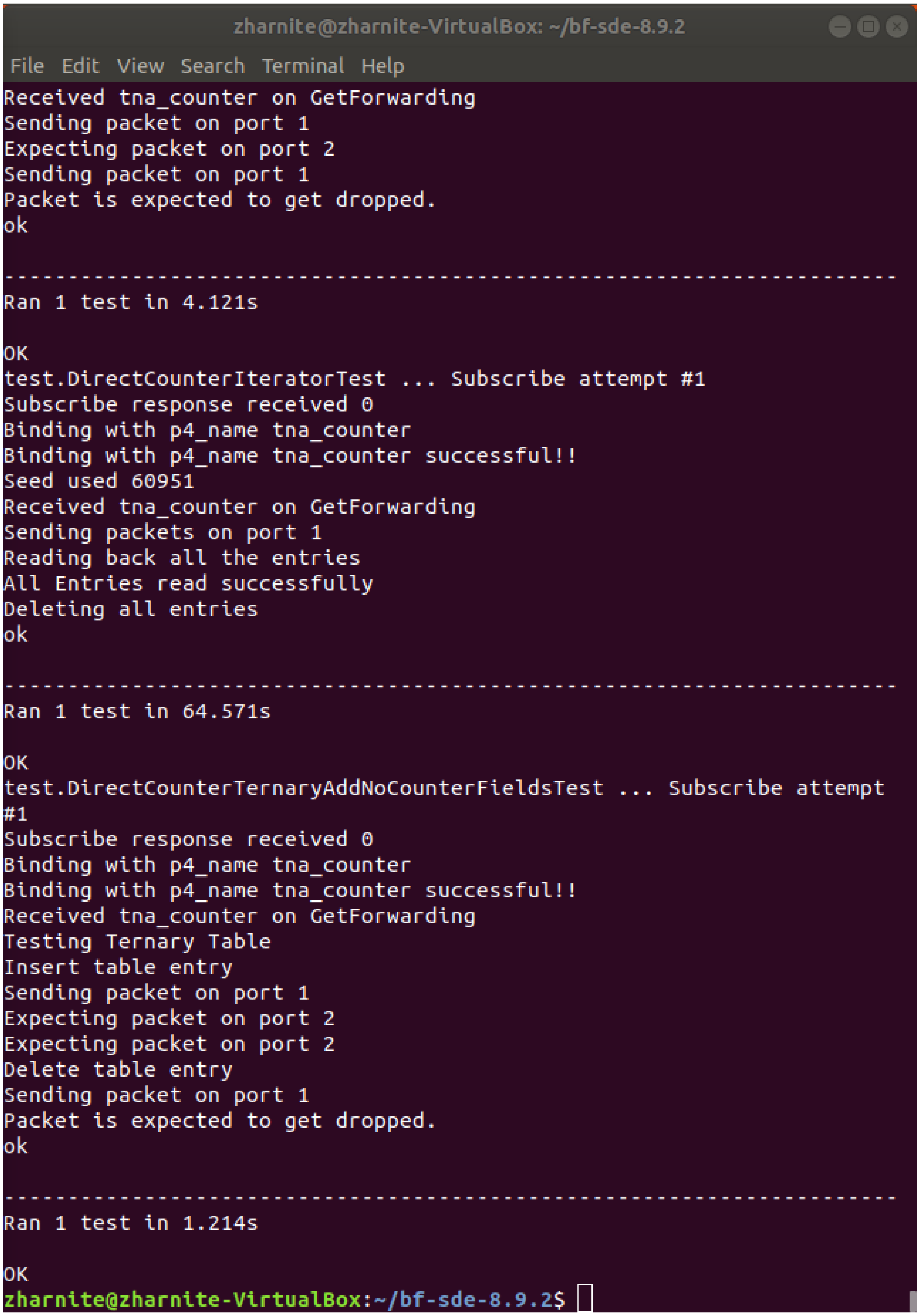
```
    libpdthrift:
    context: /home/zharnite/bf-sde-8.9.2/install/share/tofinopd/tn
a_counter/pipe/context.json
    config: /home/zharnite/bf-sde-8.9.2/install/share/tofinopd/tna
_counter/pipe/tofino.bin
  Pipes in scope [0 1 2 3 ]
  diag:
  mavericks diag:
  Agent[0]: /home/zharnite/bf-sde-8.9.2/install/lib/libpltfm_mgr.s
o
  non_default_port_ppgs: 0
  SAI default initialize: 1
Operational mode set to default: MODEL
bf_switchd: initializing dru_sim service
bf_switchd: library libdru_sim.so loaded
INFO: DRU sim MTI initialized successfully
dru_sim: client socket created
dru_sim: connected on port 8001
dru_sim: listen socket created
dru_sim: bind done on port 8002, listening...
dru_sim: waiting for incoming connections...
dru_sim: connection accepted on port 8002
dru_sim: DRU simulator running
Starting PD-API RPC server on port 9090
bf_switchd: drivers initialized
Setting core_pll_ctrl0=cd44cbfe
\
bf_switchd: dev_id 0 initialized

bf_switchd: initialized 1 devices
bf_switchd: spawning cli server thread
bf_switchd: spawning driver shell
bf_switchd: server started - listening on port 9999

        ********************************************
        *      WARNING: Authorised Access Only     *
        ********************************************

bfshell> 2020-07-24 23:39:37.108131 BF_BFRT ERROR - tableUsageGet:
340 SwitchIngress.indirect_counter Not supported
2020-07-24 23:39:37.147095 BF_BFRT ERROR - tableDefaultEntryGet:24
7 SwitchIngress.indirect_counter ERROR : Table default entry get n
ot supported
```

Figure 6. Bfshell control plane on the virtual machine.


zharnite@zharnite-VirtualBox: ~/bf-sde-8.9.2
File Edit View Search Terminal Help

```
Received tna_counter on GetForwarding
Sending packet on port 1
Expecting packet on port 2
Sending packet on port 1
Packet is expected to get dropped.
ok

----------------------------------------------------------------------
Ran 1 test in 4.121s

OK
test.DirectCounterIteratorTest ... Subscribe attempt #1
Subscribe response received 0
Binding with p4_name tna_counter
Binding with p4_name tna_counter successful!!
Seed used 60951
Received tna_counter on GetForwarding
Sending packets on port 1
Reading back all the entries
All Entries read successfully
Deleting all entries
ok

----------------------------------------------------------------------
Ran 1 test in 64.571s

OK
test.DirectCounterTernaryAddNoCounterFieldsTest ... Subscribe attempt
#1
Subscribe response received 0
Binding with p4_name tna_counter
Binding with p4_name tna_counter successful!!
Received tna_counter on GetForwarding
Testing Ternary Table
Insert table entry
Sending packet on port 1
Expecting packet on port 2
Expecting packet on port 2
Delete table entry
Sending packet on port 1
Packet is expected to get dropped.
ok

----------------------------------------------------------------------
Ran 1 test in 1.214s

OK
zharnite@zharnite-VirtualBox:~/bf-sde-8.9.2$ 
```

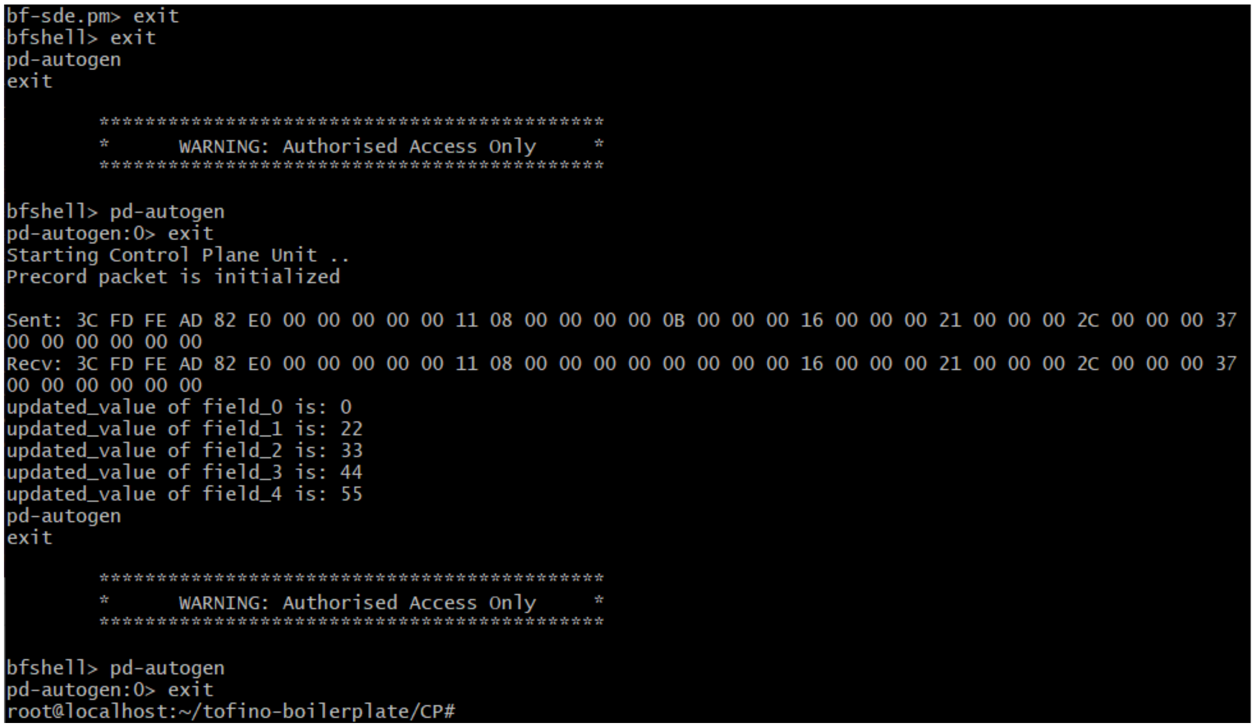
Figure 7. Python test cases running successfully for tna_counter example.

4.2 Installation on the ONIE switch.

As for the installation on the physical hardware, there still exist bugs in the software and the tofino-model is only half functioning. The errors included the non-existent linux headers, thrift dependency compile error, problems with loading and unloading the bf-drivers, a TThreadedServer exception printing every second on the bfshell control plane of the tofino-model, a platform error after installing a required driver, and many more, which were unrecorded.

4.3 Experience with the NYU device.

The NYU Tofino device is another switch that was already set up. Within this machine, there was a tofino-boilerplate written and ready to be tested. This program sends a packet from the C language control plane on a separate thread, given 5 input fields. It then receives the previously sent packet by joining the thread and printing the updated fields to the observer. A running example of this is presented in Figure 8. The commands were: "./run.sh 11 22 33 44 55." As seen from the image, the first field, originally containing 11, was updated to 0 after being received. The current goal of the project is to get Rutgers' device to a similar state.



```
bf-sde.pm> exit
bfshell> exit
pd-autogen
exit

        ********************************************
        *      WARNING: Authorised Access Only     *
        ********************************************

bfshell> pd-autogen
pd-autogen:0> exit
Starting Control Plane Unit ..
Precord packet is initialized

Sent: 3C FD FE AD 82 E0 00 00 00 00 00 11 08 00 00 00 00 0B 00 00 00 16 00 00 00 21 00 00 00 2C 00 00 00 37
00 00 00 00 00 00
Recv: 3C FD FE AD 82 E0 00 00 00 00 00 11 08 00 00 00 00 00 00 00 00 16 00 00 00 21 00 00 00 2C 00 00 00 37
00 00 00 00 00 00
updated_value of field_0 is: 0
updated_value of field_1 is: 22
updated_value of field_2 is: 33
updated_value of field_3 is: 44
updated_value of field_4 is: 55
pd-autogen
exit

        ********************************************
        *      WARNING: Authorised Access Only     *
        ********************************************

bfshell> pd-autogen
pd-autogen:0> exit
root@localhost:~/tofino-boilerplate/CP#
```

Figure 8. Running the tofino-boilerplate example.

5. DISCUSSION.

Here are some reflections on the work I have done.

5.1 Understanding the forwarding plane and control plane.

In my experience, the hardest part of the Github tutorials [3, 4] was grasping the concept of the control plane. I could not understand what the purpose of the control plane was and what was meant by populating the tables. I believed the tables had already existed on the forwarding plane and the control plane was not needed if the tables were already there. Now that I do understand the purpose of the control plane, it is hard to explain what I was previously confused about. Working through the exercises and looking at the Python code, attempting to understand it, allowed me to grasp the purpose of the two different planes.

5.2 Remembering the Barefoot SDE.

Regarding the virtual machine, it was the most gratifying experience to see a test case being run and succeeding after hours of debugging during the installation period. I was able to explore the bfshell and navigate to a Python based control plane of the program. There, I populated the tables with my custom entries and ran a tailored test case to see if the execution would pass. It did.

I encountered many problems while installing the SDE on the virtual machine; however, I will not get into the details in the interest of time and for the sake of my sanity.

6. CONCLUSION.

This project is only at the beginning of its many stages. Setting up the Barefoot SDE is the very first step to enable cutting-edge computer networking research on the university campus. There is a long way to go and it looks hopeful going forward. I wish to stay with the team and continue to contribute to the project in any way possible.

7. ACKNOWLEDGEMENTS.

I want to thank my mentor Srinivas Narayana for guiding me through this entire project and for being an amazing mentor. I want to thank the NSF grant: OAC-1925482 for providing me with the funding to do this research. I want to thank our collaborators at NYU (Professor Anirudh Sivaraman) for letting us use their Tofino device. I also want to thank the 2020 DIMACS REU program for making this magnificent experience possible. And of course, Professor Kevin McDonnell and Professor Richard McKenna for writing my letters of recommendation. Thank you all very much.

8. REFERENCES.

[1] Mckeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., . . . Turner, J. (2008). OpenFlow. ACM SIGCOMM Computer Communication Review, 38(2), 69-74. doi:10.1145/1355734.1355746

[2] Bosshart, P., Daly, D., Gibb, G., Izzard, M., Mckeown, N., Rexford, J., . . . Walker, D. (2014). P4. ACM SIGCOMM Computer Communication Review, 44(3), 87-95. doi:10.1145/2656877.2656890

[3] P4lang. (n.d.). P4lang/tutorials. Retrieved July 24, 2020, from https://github.com/p4lang/tutorials
[4] Nsg-Ethz. (n.d.). Nsg-ethz/p4-learning. Retrieved July 24, 2020, from https://github.com/nsg-ethz/p4-learning